

# Old query\_engine.hpp

---

This document is about how queries work once they are constructed. There is not yet any documentation on how they are constructed.

## What is a query?

Currently queries only work on one single table. Joins or self joins are not supported. A query consists of two things:

- One or more **conditions** (equal, greater, less) belonging to each their **condition column**
- An **action** (find, count, find\_all), optionally belonging to an **aggregate column** that works as value source (sum, max)

The conditions can be of various kinds:

- a) Requiring linear search
- b) Lookup in indexed column
- c) Lookup in string enum
- d) ListView filter (search only in rows in a given ListView – for sub queries)
- e) Searching in subtable of a column

The conditions can be related to eachother in multiple ways:

- AND
- OR
- Parantheses (begin\_group() and end\_group())

Examples of conditions are:

- greater
- less
- less\_equal
- begins\_with (strings)
- ends\_with (strings)

We have various aggregate actions

- find first
- find all
- count
- sum
- max
- min
- average

And finally various column types:

- String (subtypes ArrayString, ArrayStringLong, ArrayBigBlobs)
- Integer
- Binary
- Date
- etc

## Scheduling

Assume we have 3 conditions of types b (indexed lookup), d (ListView filter) and a (linear search). Row 0 is left and the dots are matches:



The query engine “travels” from left to right in a single pass, inside a loop belonging to the condition it currently finds is fastest. It jumps frequently between the condition loops.

When a match is found in the current condition loop, the other conditions are probed at that match index, and the search resumes.

We must first note that a linear searches and indexed lookups can outperform eachother mutually, depending on match distance and bitwidth. Especially, an indexed search that matches row 1, 11, 12, 14, 25, 30, 54 is outperformed by a linear boolean search that can test row 0 - 63 in one operation.

So we introduce two statistics variables for each condition:

**float m\_dD:**

Average distance (in rowcount) between matches around current position

**float m\_dT:**

Time (in arbitrary units) it takes to test row  $n + 1$  for a match if row  $n$  has just been tested. It depends on the condition kind:

```
Linear search:          m_dT = (bitwidth == 0 ? 1.0 / MAX_LIST_SIZE : bitwidth / 8.0)
Index, ListView and Enum: m_dT = 0
```

We also define a cost function for each condition:

```
float cost():  1.0/16.0 * m_dD + m_dT
```

The **cost** function is an expression of the **average time spent per table source row** when inside the given condition loop. Examples:

Kind	Match distance	Bit width	cost
Linear search	64	1	0.1875
Index/ListView/Enum	64	any	0.1875
Linear search	20	4	1.3
Linear search	infinity (no matches)	0	0.001
Index/ListView/Enum	15	any	1.14
Index/ListView/Enum	100	any	0.16

## Call flow

Imagine a query like **first.greater(4.56).second.not\_equal("hello").third.equal(true)** on float, string and bool columns. This is constructed as condition objects ("nodes") from various templated classes defined in query\_engine.hpp:

<Greater, float> FloatDoubleNode  
m\_value = 4.56

<Equal> IntegerNode  
m\_value = true

<NotEqual> StringNode  
m\_value = "hello"

The classes inherit from ParentNode which has common methods and variables. Most important are:

### Common methods inherited from class ParentNode:

```
std::vector<ParentNode*>m_children; // List of pointers to all other nodes so that they can call methods on eachother
size_t m_condition_column_idx;      // Column of search criteria
double m_dD;
double m_dT;
T m_value;                          // Search value
```

```
// Main entry point of a query. Can be called on any of the nodes; yields same result. Schedules calls to
// aggregate_local (see below).
```

```
// Return value is the result of the query, or Array pointer for FindAll.
```

```
template<Action TAction, class TResult, class TSourceColumn>
```

```
TReturnType aggregate(QueryState<TReturnType>* st, size_t start, size_t end, size_t agg_col, size_t* matchcount)
```

```
// Executes start..end range of a query and will stay inside the condition loop of the node it was called on. Can be
// called on any node; yields same result, but different performance. Returns prematurely if condition of
// called node has been evaluated to true local_matches number of times.
```

```
// Return value is the next row for resuming aggregating (next row that caller must call aggregate_local on)
```

```
template<Action TAction, class TResult, class TSourceColumn>
```

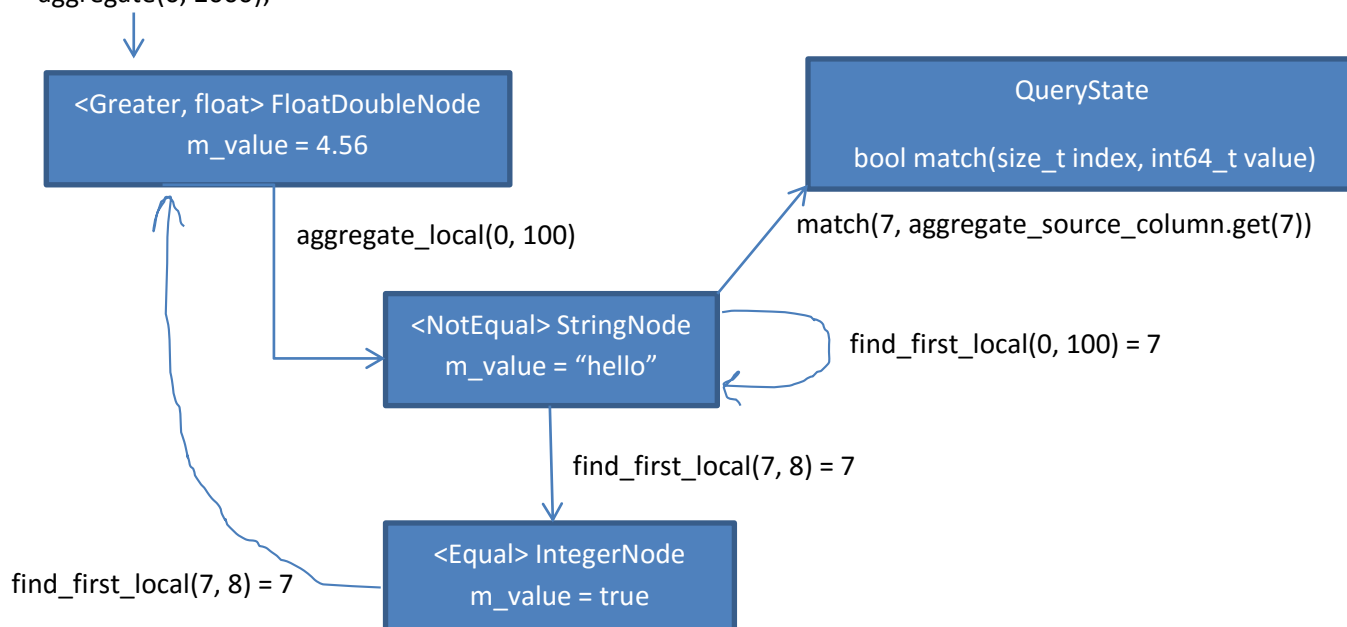
```
size_t aggregate_local(QueryState* st, size_t start, size_t end, size_t local_limit, SequentialGetter* source_column)
```

### Pure virtual method (each node class must define their own):

```
// Finds the first row (in start..end) that fulfills the condition of the node it was called on
size_t find_first_local(size_t start, size_t end)
```

There also exists a "match consuming" QueryState object. An example of call flow where row 7 is a match is given below. It is pseudo code where we only show start and end arguments for simplicity.

aggregate(0, 2000);



Pseudo code:

```
class ParentNode {

aggregate(start, end) {
    while(start < end) {
        best_node = analyze which is the best node
        start = best_node->aggregate_local(subrange_start, subrange_end);
    }

    // This is often overloaded for performance, e.g. for IntegerNode
    aggregate_local(start, end) {
        for(;;) {
            start = find_first_local(start, end);
            if(start == not_found)
                return end;

            For each other node, test if other_node->find_first_local(start, start + 1)
            is a match, and if they are, then call:
                match(start, aggregate_column->get(start));
        }
    }
}
```

We have a node class for each column type:

```
class ParentNode
template <class TConditionValue, class TConditionFunction> class IntegerNode: public ParentNode
class ListviewNode: public ParentNode
class SubtableNode: public ParentNode
template <class TConditionFunction> class StringNode: public ParentNode
template<> class StringNode<Equal>: public ParentNode
template <class TConditionValue, class TConditionFunction> class FloatDoubleNode: public ParentNode
template <class TConditionFunction> class BinaryNode: public ParentNode
class OrNode: public ParentNode
template <class TConditionValue, class TConditionFunction> class TwoColumnsNode: public ParentNode
class ExpressionNode: public ParentNode
```

## New datatype or condition?

Super simple! Make a class that provides at least following methods:

```
NewTypeNode(size_t column) {  
}  
  
void init(const Table& table) {  
}  
  
size_t find_first_local(size_t start, size_t end) {  
}
```


For performance, also overload `aggregate_local()`, but this is optional.

## IntegerNode

For conditions on integer columns (int, date, bool, etc) we utilize SSE to search in 128 bits (16 bytes) at a time, for both equal, notequal, greater, less, etc.

Assume we have an Array object of byte sized integers and perform a `find_all()`. The stars are matches. We could first call `Array::find(0, -1)`:


0				5		7				12	15					23	25					31					39				
				*						*							*									*					



SSE will first compare index 0...15 and return 5.

To find the next match, we could call `Array::find(6, -1)`:

0				5		7				12	15					23	25					31						39			
				*						*							*											*			



This requires a manual search loop for the unaligned part, because SSE can only read at aligned data (SSE 4.2 introduced a nonaligned load instruction, but its performance has not yet been tested).

This is slow and also performs redundant work, especially when match density is high.

To solve that, we have extended `Array::find()`; it can call `match()` directly for each match, provided that 1) the query has no other conditions, and 2) aggregate column (if any) is the same as condition column.

`Array::find()` can also call a callback method if there are other conditions that need to be tested.

So inside the `IntegerNode` class, we have overloaded `aggregate_local` to following:

**pseudo code**

```
class IntegerNode {

template <Action TAction, class TSourceColumn> bool match_callback(int64_t v) {
    // v is row index
    Test remaining conditions in m_children with other_node->find_first_local(v, v + 1)
    and call match(v, aggregate_column.get(v)) if all conditions are true for row v
}

// Overloading of general aggregate_local described in previous chapter
size_t aggregate_local(start, end) {
    if(there are no other conditions than us, and aggregate column is the same as ours) {
        m_array.find<TAction>(start, end, NULL, &query_state); // query_state has the match() method
    }
    else {
        m_array.find<act_CallbackIdx>(start, end, &match_callback, &query_state); //query_state contains match()
    }
}
```

## Query State

The QueryState is defined in array.hpp

It consumes matches and performs aggregate actions on them. Different actions use either index, value or nothing:

Action	Uses index	Uses value
find	+	
find_all	+	
count		
average		+
max		+
min		+
sum		+

It has the method:

```
template <Action action, bool pattern> inline bool match(size_t index, uint64_t indexpattern, int64_t value)
```

If value or index are unused by the action, the caller will most often just pass 0 for that argument (compilers are bad at detecting that the return value of get() is unused and that it has no side effects and that it can thus be elided).

Both SSE and some “integer bithacks” we have developed are very fast at generating a bit-pattern of matching rows. You can see an example of that in Array::FindGTLT\_Fast() in array.hpp.

These bitpatterns can be utilized to provide a whole chunk of matches to the QueryState at once in a single call. Set the ‘bool pattern’ template argument to true and provide it as ‘indexpattern’ argument.

The match() can decide whether or not the indexpattern is sufficient information for the given action and return ‘false’ if it rejects it (e.g. for the max/min actions). If match returns false, you must call manually one time per match with normal index and value arguments:

```
uint64_t matchpattern = BitHacks(data_chunk);
bool accepted = match<Taction, true>(0, matchpattern, 0);

if(!accepted) {
    for(int I = 0; I < 64; I++)
        ...
        extract index and values for each match
        match<Taction, false>(index, 0, value);
}
```

All this is reduced at compile time.

# New query\_expression.hpp

---

The new query syntax lets you create a query\_engine “condition node” object (as described above) which contains a syntax tree of any given expression, such as `table.first > table.second / 2 + 100`.

As described above, all you need to add new kinds of nodes is to implement a constructor, `init()` and `find_first_local`:

```
class ExpressionNode: public ParentNode {
public:
    ~ExpressionNode() TIGHTDB_NOEXCEPT
    {
        if(m_auto_delete)
            delete m_compare, m_compare = NULL;
    }

    ExpressionNode(Expression* compare, bool auto_delete)
    {
        m_auto_delete = auto_delete;
        m_child = 0;
        m_compare = compare;
    }

    void init(const Table& table)
    {
        m_compare->set_table(&table);
        if (m_child)
            m_child->init(table);
    }

    size_t find_first_local(size_t start, size_t end)
    {
        size_t res = m_compare->find_first(start, end);
        return res;
    }

    bool m_auto_delete;
    Expression* m_compare; < query_expression expression tree object
};
```

This allows the user to write things like:

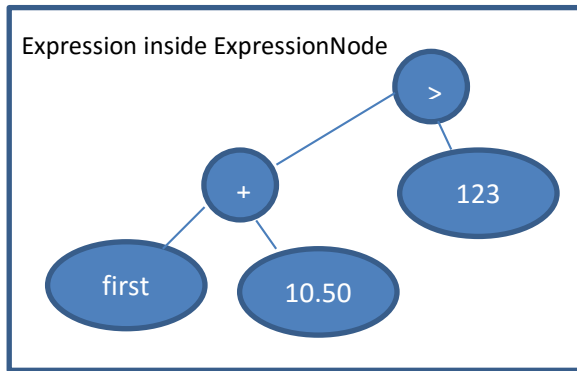
```
size_t t = (table.first > table.second / 2 + 100).find();
```

where right hand side is a normal **Query object with .find(), .max(), etc.**



## Expression type (m\_compare above)

An Expression consists of **one** condition such as >, ==, !=, etc (it could be an idea to rename it into Condition instead):



If a query contains multiple conditions delimited by && or ||, they are

Type conversion/promotion semantics is the same as in the C++ expressions, e.g `float + int > double == float + (float)int > double`.

Grammar:

---

```
Expression:      Subexpr2<T>  Compare<Cond, T>  Subexpr2<T>

Subexpr2<T>:     Value<T>
                  Columns<T>
                  Subexpr2<T> Operator<Oper<T>> Subexpr2<T>
                  power(Subexpr2<T>) // power(x) = x * x, as example of unary operator

Value<T>:        T

Operator<Oper<T>>: +, -, *, /

Compare<Cond, T>: ==, !=, >=, <=, >, <

T:               bool, int, int64_t, float, double, StringData
```

Class diagram

---

```
Subexpr2
    void evaluate(size_t i, ValueBase* destination)

Compare: public Subexpr2
    size_t find_first(size_t start, size_t end)    // main method that executes query

    bool m_auto_delete
    Subexpr2& m_left;                             // left expression subtree
    Subexpr2& m_right;                             // right expression subtree

Operator: public Subexpr2
    void evaluate(size_t i, ValueBase* destination)
    bool m_auto_delete
    Subexpr2& m_left;                             // left expression subtree
    Subexpr2& m_right;                             // right expression subtree

Value<T>: public Subexpr2
    void evaluate(size_t i, ValueBase* destination)
    T m_v[8];

Columns<T>: public Subexpr2
    void evaluate(size_t i, ValueBase* destination)
    SequentialGetter<T> sg;                       // class bound to a column, lets you read values in a fast way
    Table* m_table;

class ColumnAccessor<>: public Columns<double>
```

Call diagram:

Example of 'table.first > 34.6 + table.second':

Operator, Value and Columns have an `evaluate(size_t i, ValueBase* destination)` method which returns a `Value<T>` containing 8 values representing table rows `i...i + 7`.

Memory allocation:

Value and Columns given to Operator or Compare constructors are cloned with 'new' and hence deleted unconditionally by query system.

## Caveats, notes and todos

---

- \* Perhaps disallow columns from two different tables in same expression
- \* The name Columns (with s) can be confusing because we also have Column (without s)
- \* Memory allocation: Maybe clone Compare and Operator to get rid of m\_auto\_delete. However, this might become bloated, with non-trivial copy constructors instead of defaults
- \* Hack: In compare operator overloads (==, !=, >, etc), Compare class is returned as Query class, resulting in object slicing. Just be aware.
- \* clone() some times new's, sometimes it just returns \*this. Can be confusing. Rename method or copy always.
- \* We have Columns::m\_table, Query::m\_table and ColumnAccessorBase::m\_table that point at the same thing, even with ColumnAccessor<> extending Columns. So m\_table is redundant, but this is in order to keep class dependencies and entanglement low so that the design is flexible (if you perhaps later want a Columns class that is not dependent on ColumnAccessor)

\*/